Automatic test suite generation for PMCFG grammars

Inari Listenmaa and Koen Claessen
Department of Computer Science and Engineering
University of Gothenburg and Chalmers University of Technology
Gothenburg, Sweden
inari@chalmers.se, koen@chalmers.se

Abstract

We present a method for finding errors in formalized natural language grammars, by automatically and systematically generating test cases that are intended to be judged by a human oracle. The method works on a per-construction basis; given a construction from the grammar, it generates a finite but complete set of test sentences (typically tens or hundreds), where that construction is used in all possible ways. Our method is an alternative to using a corpus or a treebank, where no such completeness guarantees can be made. The method is language-independent and is implemented for the grammar formalism PMCFG, but also works for weaker grammar formalisms. We evaluate the method on a number of different grammars for different natural languages.

1 Introduction

What is the *essence* of a language? When formalizing and implementing a natural language grammar, which example sentences do we need to check in order to convince ourselves that the grammar is correct?

Imagine we are formalizing a grammar for English, and in particular we are working on the reflexive construct. In order to check correctness for the 3rd person singular, we need to test for three different subjects, because the object has to agree with the subject: "he sees himself", "she sees herself" and "it sees itself". Without seeing all three examples, we cannot be certain that the reflexive construction is implemented correctly. In contrast, the general pattern of a transitive verb with a non-reflexive object is enough to test with only one third person subject: *he, she, it*, or any singular noun or proper name. The agreement only shows in the verb form, thus including both "she sees a dog" and "John sees a dog" in the test suite is redundant.

What is minimal and representative depends on the language. For instance, Basque transitive verbs agree with both subject and object, thus we need 6×6 examples just to cover all verb forms. In this paper, we are not interested in the morphology per se—there are easier methods to test for that—but the correctness of the syntactic function: does the function pick the correct verb form for the correct combination of subject and object? For that purpose, it is enough to test the syntactic construction "transitive verb phrase" with just a single transitive verb.

We present a method that, given a grammar (that in general encompasses an infinite set of sentences), generates a finite set of sentences that can be used as test cases for the correctness of the grammar. Our concrete implementation is for a particular grammar formalism, namely parallel multiple context-free grammars (PMCFG) (Seki et al., 1991), which is the core formalism used by the Grammatical Framework (GF) (Ranta, 2004). However, the general method works for any formalism that is at most as expressive as PMCFG: context-free grammars (CFG) and multiple context-free grammars (MCFG), which covers formalisms such as Tree-Adjoining Grammar (Joshi et al., 1975), Combinatorial Categorial Grammar (Steedman, 1988) and Minimalist Grammar (Stabler, 1997).

2 Grammatical Framework

Grammatical Framework (GF) (Ranta, 2004) is a framework for building multilingual grammar applications. Its main components are a functional programming language for writing grammars and a resource library (Ranta, 2009), which, as of May 2018, contains the linguistic details of 40 natural languages. GF is well suited for tasks where quality matters more than coverage—examples include mathematical and legal domains (Caprotti, 2006; Camilleri, 2017). The library has had over 50 contributors, and it consists of 1900 program modules and 3 million lines of code! The complexity of such a piece of code clearly motivates dedicated test methods, such as the one described in this paper.

A GF grammar consists of an *abstract syntax*, which is a set of grammatical categories and functions between them, and one or more *concrete syntaxes*, which describe how the abstract functions and categories are linearized, i.e. turned into surface strings. The resulting grammar describes a mapping between concrete language strings and their corresponding abstract trees. This mapping is bidirectional—strings can be *parsed* to trees, and trees *linearized* to strings. As an abstract syntax can have multiple corresponding concrete syntaxes, the respective languages can be automatically *translated* from one to the other by first parsing a string into a tree and then linearizing the obtained tree into a new string. Figure 1 shows the abstract syntax of a small example grammar in GF, slightly modified from (Ranta, 2011), and Figure 2 shows a corresponding Spanish concrete syntax. We refer to this grammar throughout sections 2–4.

```
abstract Foods = {
  flags startcat = S ;
  cat
    S ; NP ; CN ; AP ;
  fun
    Pred : NP -> AP -> S ;                    -- this wine is good
    This, That, These, Those : CN -> NP ;     -- this wine
    Mod : AP -> CN -> CN ;                    -- Italian wine
    Wine, Cheese, Fish, Pizza : CN ;
    Warm, Good, Italian, Vegan : AP ;
```

Figure 1: Abstract syntax of a GF grammar about food

2.1 Abstract syntax

Abstract syntax describes the constructions in our grammar without giving a concrete implementation. Section `cat` introduces the categories of the grammar: `S` for sentence, `NP` for noun phrase, `AP` for adjectival phrase, and `CN` for common noun, which can be modified by `AP`s, but hasn't yet been quantified into an `NP`. `S` is the *start category* of the grammar: this means that sentences are complete constructions in the language, everything else is an intermediate stage.

Section `fun` introduces functions: they are either lexical items without arguments, or syntactic functions which manipulate their arguments and build new terms. Of the syntactic functions, `Pred` constructs an `S` from an `NP` and an `AP`, building trees such as `Pred (This Pizza) Good` 'this pizza is good'. `Mod` adds an `AP` to a `CN`, such as `Mod Italian Pizza` 'italian pizza'. The functions `This, That, These` and `Those` quantify a `CN` into an `NP`, for instance, `That (Mod Italian Pizza)` 'that italian pizza'.

2.2 Concrete syntax

Concrete syntax is an implementation of the abstract syntax. The section `lincat` corresponds to `cat` in the abstract syntax: for every abstract category introduced in `cat`, we give a concrete implementation in `lincat`.

```
concrete FoodsSpa of Foods = {
  lincat
    S = Str ;
    NP = { s : Str ; n : Number ; g : Gender } ;
    CN = { s : Number => Str ; g : Gender } ;
    AP = { s : Number => Gender => Str ; p : Position } ;
  lin
    Pred np ap = np.s ++ copula ! np.n ++ ap.s ! np.n ! np.g ;
    This cn = mkNP Sg "este" "esta" cn ;
    These cn = mkNP Pl "estos" "estas" cn ;
    -- That, Those defined similarly
    Mod ap cn = { s = \\n => preOrPost ap.p (ap.s ! n ! cn.g) (cn.s ! n) ;
                  g = cn.g } ;
    --Wine, Cheese, ..., Italian, Vegan defined as lexical items
  param
    Number = Sg | Pl ;
    Gender = Masc | Fem ;
    Position = Pre | Post ;
  oper
    mkNP num mascDet femDet cn =
     let det = case cn.g of { Masc => mascDet ; Fem => femDet } ;
      in { s = det ++ cn.s ! num ; n = num ; g = cn.g } ;
    copula = table { Sg => "es" ; Pl => "son" } ;
    preOrPost p x y = case p of { Pre => x ++ y ; Post => y ++ x } ;
```

Figure 2: Spanish concrete syntax of a GF grammar about food

Figure 2 shows the Spanish concrete syntax, in which `S` is a string, and the rest of the categories are more complex records. For instance, `CN` has a field `s` which is a table from number to string (SG ⇒ *pizza*, PL ⇒ *pizzas*), and another field `g`, which contains its gender (feminine for `Pizza`). We say that `CN` has *inherent* gender, and *variable* number.

The section `lin` contains the concrete implementation of the functions, introduced in `fun`. Here we handle language-specific details such as agreement: when `Pred (This Pizza) Good` is linearized in Spanish, 'esta pizza es buena', the copula must be singular (*es* instead of plural *son*), and the adjective must be in singular feminine (*buena* instead of masculine *bueno* or plural *buenas*), matching the gender of `Pizza` and the number of `This`. If we write an English concrete syntax, then only the number of the copula is relevant: this pizza/wine *is* good, these pizzas/wines *are* good.

3 PMCFG

GF grammars are compiled into parallel multiple context-free grammars (PMCFG), which are processed by our tool. Here we explain three key features for the test suite generation.

3.1 Concrete categories

For each category in the original grammar, the GF compiler introduces a new *concrete category* in the PMCFG for each combination of inherent parameters. These concrete categories can be linearized to strings or vectors of strings. The start category (`S` in the Foods grammar) is in general a single string, but intermediate categories may have to keep several options open.

Consider the categories `NP`, `CN` and `AP` in the Spanish concrete syntax. Firstly, `NP` has inherent number and gender, so it compiles into four concrete categories: $\texttt{NP}_{\texttt{sg}\times\texttt{masc}}$, $\texttt{NP}_{\texttt{sg}\times\texttt{fem}}$, $\texttt{NP}_{\texttt{pl}\times\texttt{masc}}$

and $\texttt{NP}_{\texttt{pl}\times\texttt{fem}}$, each of them containing one string. Secondly, CN has an inherent gender and variable number, so it compiles into two concrete categories: $\texttt{CN}_{\texttt{masc}}$ and $\texttt{CN}_{\texttt{fem}}$, each of them a vector of two strings (singular and plural). Finally, AP needs to agree in number and gender with its head, but it has its position as an inherent feature. Thus AP compiles into two concrete categories: $\texttt{AP}_{\texttt{pre}}$ and $\texttt{AP}_{\texttt{post}}$, each of them a vector of four strings.

3.2 Concrete functions

Just like categories, each syntactic function from the original grammar turns into multiple syntactic functions in the PMCFG—one for each combination of parameters of its arguments.

- $\texttt{Mod}_{\texttt{pre}\times\texttt{fem}} : \texttt{AP}_{\texttt{pre}} \to \texttt{CN}_{\texttt{fem}} \to \texttt{CN}_{\texttt{fem}}$
- $\texttt{Mod}_{\texttt{post}\times\texttt{fem}} : \texttt{AP}_{\texttt{post}} \to \texttt{CN}_{\texttt{fem}} \to \texttt{CN}_{\texttt{fem}}$
- $\texttt{Mod}_{\texttt{pre}\times\texttt{masc}} : \texttt{AP}_{\texttt{pre}} \to \texttt{CN}_{\texttt{masc}} \to \texttt{CN}_{\texttt{masc}}$
- $\texttt{Mod}_{\texttt{post}\times\texttt{masc}} : \texttt{AP}_{\texttt{post}} \to \texttt{CN}_{\texttt{masc}} \to \texttt{CN}_{\texttt{masc}}$

3.3 Coercions

As we have seen, AP in Spanish compiles into $\texttt{AP}_{\texttt{pre}}$ and $\texttt{AP}_{\texttt{post}}$. However, the difference of position is meaningful only when the adjective is modifying the noun: "la *buena* pizza" vs. "la pizza *vegana*". But when we use an adjective in a predicative position, both classes of adjectives behave the same: "la pizza es *buena*" and "la pizza es *vegana*". As an optimization strategy, the grammar creates a *coercion*: both $\texttt{AP}_{\texttt{pre}}$ and $\texttt{AP}_{\texttt{post}}$ may be treated as $\texttt{AP}_*$ when the distinction doesn't matter. Furthermore, the function $\texttt{Pred} : \texttt{NP} \to \texttt{AP} \to \texttt{S}$ uses the coerced category $\texttt{AP}_*$ as its second argument, and thus expands only into 4 variants, despite there being 8 combinations of $\texttt{NP}\times\texttt{AP}$.

- $\texttt{Pred}_{\texttt{sg}\times\texttt{fem}\times*} : \texttt{NP}_{\texttt{sg}\times\texttt{fem}} \to \texttt{AP}_* \to \texttt{S}$
- $\texttt{Pred}_{\texttt{pl}\times\texttt{fem}\times*} : \texttt{NP}_{\texttt{pl}\times\texttt{fem}} \to \texttt{AP}_* \to \texttt{S}$
- $\texttt{Pred}_{\texttt{sg}\times\texttt{masc}\times*} : \texttt{NP}_{\texttt{sg}\times\texttt{masc}} \to \texttt{AP}_* \to \texttt{S}$
- $\texttt{Pred}_{\texttt{pl}\times\texttt{masc}\times*} : \texttt{NP}_{\texttt{pl}\times\texttt{masc}} \to \texttt{AP}_* \to \texttt{S}$

4 Generating the test suite

We now have all building blocks for creating a representative and minimal set of test cases. In the previous section, we saw how a single abstract category compiles into multiple concrete categories, depending on the combinations of parameters. Instead of dealing with parameters directly, we now have a set of new *types*, which is helpful for generating test cases.

We use one syntactic function as the base for one set of test cases. If the parameter set-up of the linearization types is are correctly chosen, the set of generated test cases will expose all bugs in the tested function.

For lexical categories, it also makes sense to test the whole category, i.e. generate all trees that show that adjectives in general are defined and handled correctly in the functions. However, we only explain in detail the method with one syntactic function as a base.

We assume that all test cases are trees with the same start (top-level) category, such as S in our example grammar. The requirement is that the start category is linearized as one string only.

4.1 Enumerate functions

As we explained before, each syntactic function turns into multiple versions, one for each combination of parameters of its arguments. We test each of these versions seperately. Each concrete syntactic function may produce one or several trees.

In order to construct trees that use the syntactic function, we need to supply the function with *arguments*, as well as put the resulting tree into a *context* that produces a tree in the correct start category.

4.2 Enumerate arguments

Some syntactic functions are simply a single lexical item (for example the word *good*); in this case just the tree `Good` is our answer. If we choose a function with arguments, such as `Pred`, then we have to supply it with argument trees. Each argument needs to be a tree belonging to the right category; in the case of `Pred`, the categories are `NP` and `AP`.

When we test a function, we want to see whether or not it uses the right information from its arguments, in the right way. The information that a syntactic function uses is any of the strings that come from linearizing its arguments. In order to be able to see which string in the result comes from which field from which argument, we want to generate test cases that only contain unique strings. For example, `Good` 'bueno/buena/buenos/buenas' is a better argument than `Warm` 'caliente/caliente/calientes/calientes', because the latter is invariable for gender. If there is no one argument that only contains unique strings, we generate a set of arguments, where for each pair of strings from the arguments, there is always one test case where those strings are different. In this way, if the syntactic function contains a mistake, there is always one test case that reveals it.

In our implementation, we use the FEAT framework (Duregård et al., 2012) to enumerate possible combinations of argument trees, in size order. We stop when we have found a suitable set of combinations of arguments, according to the requirement above. We also stop (give up) when too many tries have been made (currently 10,000), but we have not seen this happen in practice.

`Mod Good Pizza` (SG) buena pizza (PL) buenas pizzas	`Mod Good Wine` (SG) buen vino (PL) buenos vinos
`Mod Vegan Pizza` (SG) pizza vegana (PL) pizzas veganas	`Mod Vegan Wine` (SG) vino vegano (PL) vinos veganos

Table 1: Agreement and placement of adjectives in attributive position

Table 1 shows test cases for the function `Mod : AP → CN → CN` in the Spanish concrete syntax. Firstly, we need a minimal and representative set of arguments: one premodifier and one postmodifier `AP` (`Good` and `Vegan`), as well as one feminine and one masculine `CN` (`Pizza` and `Wine`). Thus our full set of test cases are `Mod` applied to the cross product of $\{^{\texttt{Good}}_{\texttt{Vegan}}\} \times \{^{\texttt{Pizza}}_{\texttt{Wine}}\}$.

4.3 Enumerate contexts

The third and last enumeration we perform when generating test cases is to generate all possible *uses* of a function. After we provide a function with arguments, we need to put the resulting tree into a context, so that we can generate a single string from the result. By *context*, we mean a tree in the start category (`S` in our example) with a *hole* of the same type as the function under test (denoted by `_`).

`Pred (This (Mod Good Pizza)) Italian` esta buena pizza es italiana	`Pred (This (Mod Good Wine)) Italian` este buen vino es italiano
`Pred (These (Mod Good Pizza)) Italian` estas buenas pizzas son italianas	`Pred (These (Mod Good Wine)) Italian` estos buenos vinos son italianos
`Pred (This (Mod Vegan Pizza)) Italian` esta pizza vegana es italiana	`Pred (This (Mod Vegan Wine)) Italian` este vino vegano es italiano
`Pred (These (Mod Vegan Pizza)) Italian` estas pizzas veganas son italianas	`Pred (These (Mod Vegan Wine)) Italian` estos vinos veganos son italianos

Table 2: Complete test cases to test `Mod`

The important thing here is that the generated set of contexts shows all the possible different ways the tree can be used. For example, for a test tree with an inflection table of 4 forms, we would generate 4 different sentences in which each of the 4 inflections is used.

As an example, consider generating contexts for a tree of category `CN`. Since `CN` is variable for number, we need two contexts: one that picks out the singular form and other that picks out the plural form. This suggests that we should apply two different `CN` $\rightarrow$ `NP` functions, for instance `This` and `These`, and give their results to the `Pred` function, which constructs an `S`. In contrast, the second argument to `Pred` doesn't make a difference in what form we pick out of the `CN`—we just want to pick something that has maximally different forms, so the program is sure not to pick `Warm`, which is invariable for gender. By random selection, let us pick `Italian`. The final two contexts are therefore `Pred (This _) Italian` and `Pred (These _) Italian`. We insert the 4 test cases from Table 1 into the 2 contexts, and get 8 trees in total, as shown in Table 2.

Fixpoint iteration In our tool, computing relevant contexts in a given start category `S`, is done once, in advance, for all possible hole types H at the same time, using a fixpoint iteration. It is possible to express the set of relevant contexts for one hole type H in terms of the sets of relevant contexts for other hole types H':

$$\mathsf{contexts}(H) = \mathsf{filter}(\ \{C[F(\_)] \mid F \in H \rightarrow H',\ C \in \mathsf{contexts}(H')\}\ )$$

In words, to generate contexts with holes of type H, we enumerate all functions F that have a H as an argument, and enumerate all contexts C that have the result type H' of F as a hole type, and put C and F together to get a new context. Then, we apply a function filter to the result in order to filter out redudant contexts, i.e. contexts whose uses of the strings of H are already covered by other contexts in the same set.

To compute relevant contexts with the start category `S` as a hole, we use the following definition $\mathsf{contexts}(\texttt{S}) = \{\ \_\ \}$. Now, in order to compute all sets of contexts for all possible hole categories H, we set up a system of equations (as specified above). In general, this system of equations is recursive, and we use a fixpoint iteration to solve it, starting with the empty set for each set of contexts. There is a guaranteed minimal solution, because the RHSs are monotonic in H'.

4.4 Pruning the trees within one set of test cases

On the scope of our tiny example grammar, this pruning method is easiest to illustrate when we test a category instead of a function; however, in bigger grammars, the need arises when testing functions as well. In order to test the category `AP`, we need in total 12 example sentences:

- 4 test cases for a premodifier `AP` as modifier;
- 4 test cases for a postmodifier `AP` as modifier;
- 4 test cases for *any* `AP` as predicative.

~~4 test cases for a premodifier `AP` as predicative~~
~~4 test cases for a postmodifier `AP` as predicative~~

These categories correspond to the concrete categories $\texttt{AP}_{\texttt{pre}}$, $\texttt{AP}_{\texttt{post}}$ and the coercion $\texttt{AP}_{*}$ (as explained in Section 3.3). However, without this pruning step, the method would generate in total 16 trees: it would insert both pre- and postmodifier `AP`s into a predicative context. With the help of coercions in the grammar, we detect and remove redundant trees from the test set.

4.5 Pruning the trees to test the whole grammar

So far we have completely ignored that one tree can showcase more than one function. In fact, the 8 test sentences created for `Mod` happen to also test `Pred` exhaustively. Let us recap the steps we took to create them for `Mod`: enumerating arguments brought us `Good`, `Vegan`, `Pizza` and `Wine`, and enumerating contexts brought us `This` and `These`. Had we been creating test cases

for `Pred`, we would've gotten `This` and `These` at the stage of enumerating arguments, and then there would've been no need for contexts, because `Pred` already creates the start category `S`.

There is a simple way to detect the redundancy: make the generation of arguments completely deterministic, e.g. always choose the function that is alphabetically first. The downside is that we get a lot of redundancy, in the style of "the pizza gives the pizza the pizza". However, if we want to generate sentences for the whole grammar at one go, we can split the generation in two stages: first stage is deterministic, where every feminine noun is `Pizza`, and we can eliminate redundancies by just eliminating copies of the same tree. Then, when we have a set of unique trees, we can substitute individual words in them with other words in the same concrete category.

It would be ideal to generate sentences that make sense, such as "the waitress gives the girl the pizza". If the grammar is purely syntactic, we would need external tools to ensure semantic coherence, but if the grammatical categories already include semantic distinctions, e.g. limiting the subject and indirect object of *give* to humans, that naturally restricts the generated test suite.

5 Example use

Let us take the GF resource grammar (Ranta, 2009) for Spanish, and pick the function `AdvCN : Adv → CN → CN`, modifies a `CN` with an adverb. The tool generates test cases in the way described previously, which include the following (slightly simplified) trees:

- `AdvCN (PrepNP next_to (DetNP your)) hill` 'hill next to yours'
- `AdvCN (PrepNP next_to (DetNP your)) house` 'house next to yours'

In Spanish, the words *hill* and *house* have different genders, and the word *yours* has to agree in gender with the antecedent: *(la) casa al lado de la tuya* and *(el) cerro al lado del tuyo*. The test cases reveal a bug, where `DetNP your` picks a gender too soon, say always masculine, instead of leaving it open in an inflection table. We implement a fix by adding gender as a parameter to the `Adv` type, and have `AdvCN` choose the correct form based on the gender of the `CN`.

After implementing the fix, we run a second test case generation, in order to compare the old and new versions of the grammar. We want to make sure that our changes have not caused new bugs in other functions. The simplest strategy is to generate test cases for *all* functions in both grammars, and only show those outputs that differ between the grammars. After our fixes, we get the following differences:

```
DetCN the (AdvCN (PrepNP next_to (DetNP your)) house)
```

- ~~*la casa al lado* **del tuyo**~~
- *la casa al lado* **de la tuya**

```
DetCN the (AdvCN (PrepNP next_to (DetNP this)) house)
```

- ~~*la casa al lado de* **este**~~
- *la casa al lado de* **esta**

We notice a side effect that we may not have thought of: the gender is retained in all adverbs made of NPs made of determiners, so now it has become impossible to say "the house next to *that*" and pointing to a hill. So we do another round of modifications, compute the difference (to the original grammar or to the intermediate), and see if something else fails.

6 Evaluation

We generate test cases for two grammars of different sizes, using three languages from different language families: Dutch, Estonian and Basque. Dutch, an Indo-European language, has fairly simple nominal and verbal morphology, but the rules for handling word order, prefixes and particles in verb phrases are somewhat intricate. Estonian and Basque both have rich noun

morphology, each featuring 14 grammatical cases, but Basque verb morphology is much more complex. Table 3 shows what happens when we ran our tool on some example functions from the resource grammar, and Table 4 shows the number of generated trees in total for all syntactic functions in two different grammars. As stated earlier, we do not consider generating test cases for all functions an optimal way of testing a whole resource grammar from scratch; Table 4 gives merely a baseline reduction from all possible trees up to a reasonable depth. We introduce the grammars and comment on the results in the following sections.

Resource grammar function	**Dutch**	**Estonian**	**Basque**
`ComparA` 'stronger than you'	10	20	6
`ComplVS` 'say that I sleep'	89	171	104
`RelNP` 'a cat that I saw'	2	23	20
`ReflVP` 'see myself'	1034	343	1082

Table 3: Test cases for some individual functions in the resource grammar

		Concrete →	**Dutch**		**Estonian**		**Basque**	
↓ Abstract	#funs+lex	#trees	#total	#uniq	#total	#uniq	#total	#uniq
Phrasebook	130+160	>480,000	513	419	610	505	538	503
Resource gr.	217+446	>500 billion	21,370	19,825	13,733	9,194	100,967	64,390

Table 4: Test cases for all functions in two grammars

6.1 Grammars

The first grammar is Phrasebook (Ranta et al., 2012), a mid-size application grammar with 42 categories such as `Person, Currency, Price` and `Nationality`, 160-word lexicon and 130 functions with arguments. The trees in the Phrasebook are more semantic than the trees we have seen so far: for example, the abstract tree for the sentence "how far is the bar?" in the Phrasebook is `PQuestion (HowFar (ThePlace Bar))`, at depth 3, in contrast to the resource grammar tree[1] for the same sentence, which is of depth 6. Limiting up to depth 3, the Phrasebook grammar produces over 480,000 trees.

The second grammar is a subset of the GF resource grammar (Ranta, 2009), with 84 categories, 217 syntactic functions and 446 words in the lexicon. Since all the languages did not have a complete implementation, we simply took the subset of functions that was common, and removed manually a couple of rare constructions and words that are potentially distracting. This fragment produces hundreds of billions of trees already up to depth 5. None of the resource grammars has been tested systematically before—for the Estonian grammar (Listenmaa and Kaljurand, 2014), the morphological paradigms were tested extensively against existing resources, but syntactic functions were only tested with a treebank of 425 trees.

6.2 Results

Execution time We ran all the experiments on a MacBook Air with 1,7 GHz processor and 8 GB RAM. For Phrasebook, it took just seconds to generate the test suite for all languages. For the resource grammar, Dutch and Estonian finished in 3–4 minutes. However, the Basque resource grammar is noticeably more complex, and creating test trees for all functions took almost three hours.

Generated trees We report both total and unique trees: total trees are simply the sum of all trees for all functions, and unique trees is the count after removing duplicates, as explained in Section 4.5 ("the pizza gives the pizza the pizza" style).

[1] `UttQS (UseQCl (TTAnt TPres ASimul) PPos (QuestIComp (CompIAdv (AdvIAdv how_IAdv (PositAdvAdj far_A))) (DetCN (DetQuant DefArt NumSg) (UseN bar_N))))`

As we can see, the number of trees differs a lot. Table 3 shows some expected patterns: Basque and Estonian have more complex nouns than Dutch, so `RelNP` needs more tests. Likewise, Basque and Dutch have more complex verb phrases (for different reasons!) than Estonian, so `ReflVP` needs more tests. In general, we believe the number of test cases has both language typological and grammar engineering reasons. Other resource grammarians have reported significant differences in complexity between implementations: (Enache et al., 2010) report a 200-time reduction in the number of concrete rules after changing the implementation of clitics in verb phrases. It would be interesting to test two different implementations of the same language using this method.

Finding bugs The Basque resource grammar is still a work in progress, and the test sentences showed serious problems in morphology. We thought it premature to get a fluent speaker to evaluate the grammar, because the errors in morphology would probably make it difficult to assess syntax separately. Phrasebook was implemented using the resource grammar, so it was equally error-ridden.

For Estonian, we read through all the Phrasebook test sentences. These 505 sentences showed a handful of bugs, all coming from Phrasebook itself not the resource grammar. Most were individual words having the wrong inflection paradigm (the right one exists in the resource grammar, but wrong one was chosen by the application grammarian), but there were also some bugs in more general functions—for example, using a wrong form of nationality when applied to a human and when to an institution, along the lines of "Spaniard restaurant". As expected, Phrasebook sentences were easier to read, and made more sense semantically than sentences from the resource grammar.

We have been developing the tool by testing it on the Dutch resource grammar, and during 6 months, we have committed 22 bugfixes on Dutch in the GF main repository. (In the name of honesty, a few of the bugs were caused by our earlier "fixes"—that was before we had implemented the comparison against an older version of the grammar, explained in Section 5.) One of the bugs found in Dutch was also present in other languages, so we fixed it in German and English.

7 Related work

Our approach is inspired by automatic test case generation for general software, such as (Hamon et al., 2004). Software testing and grammar testing both deal with notions of coverage and compactness. In computational linguistics, there has been work on error mining for parsing grammars, such as (Gardent and Narayan, 2012).

There is a long tradition of hand-crafted test suites for various grammar formalisms. Traditionally, these test suites include also ungrammatical sentences: those that the grammar should *not* be able to parse. However, most of the previous work is meant for monolingual grammars which are either parsing or generation grammars, whereas GF is a reversible and multilingual formalism: thus it is natural to represent test cases as trees. Finally, our method generalizes to many kinds of grammars, including application grammars where the distinctions are not syntactic but semantic.

8 Conclusion and future work

We have presented method for automatically generating minimal and exhaustive sets of test cases for testing grammars. We have found the tool useful in large-scale grammar writing, in a context where grammars need to be *reliable*.

One problem we have encountered is that the test sentences from resource grammars are often nonsensical semantically, and hence a native speaker might intuitively say that a sentence is wrong, even though it is just unnatural. For instance, the function `AdvQVP` covers constructions such as "you did *what*?". However, the function itself is completely general and can take any verb phrase and any question adverb, thus bizarre combinations like "you saw the dog why" may appear in the generated test cases. For future work, we plan to use external resources to guide the algorithm to pick trees that also make sense semantically.

So far the only mode of operation is generating test cases for a single function. As future work, we are planning to add a separate mode for testing the whole grammar from scratch: intentionally create trees that test several functions at once. We have an implementation only for GF grammars so far, but the general method works for any grammar formalism that can be compiled into PMCFG. GF already supports reading context-free grammars, so testing any existing CFG is a matter of some preprocessing.

References

[Camilleri2017] John J. Camilleri. 2017. *Contracts and Computation–Formal modelling and analysis for normative natural language*. Ph.D. thesis, University of Gothenburg, Gothenburg, Sweden.

[Caprotti2006] Olga Caprotti. 2006. WebALT! Deliver mathematics everywhere. In *Society for Information Technology & Teacher Education International Conference*, pages 2164–2168. Association for the Advancement of Computing in Education (AACE).

[Duregård et al.2012] Jonas Duregård, Patrik Jansson, and Meng Wang. 2012. FEAT: Functional Enumeration of Algebraic Types. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, pages 61–72, New York, NY, USA. ACM.

[Enache et al.2010] Ramona Enache, Aarne Ranta, and Krasimir Angelov. 2010. An open-source computational grammar for Romanian. In *International Conference on Intelligent Text Processing and Computational Linguistics*, pages 163–174. Springer.

[Gardent and Narayan2012] Claire Gardent and Shashi Narayan. 2012. Error mining on dependency trees. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Long Papers - Volume 1*, ACL '12, pages 592–600, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Hamon et al.2004] Grégoire Hamon, Leonardo de Moura, and John Rushby. 2004. Generating efficient test sets with a model checker. In *Proceedings of the Software Engineering and Formal Methods, Second International Conference*, SEFM '04, pages 261–270, Washington, DC, USA. IEEE Computer Society.

[Joshi et al.1975] Aravind K. Joshi, Leon S. Levy, and Masako Takahashi. 1975. Tree Adjunct Grammars. *Journal of Computer and System Sciences*, 10(1):136–163.

[Listenmaa and Kaljurand2014] Inari Listenmaa and Kaarel Kaljurand. 2014. Computational Estonian Grammar in Grammatical Framework. In *9th SALTMIL workshop on free/open-source language resources for the machine translation of less-resourced languages*.

[Ranta et al.2012] Aarne Ranta, Ramona Enache, and Grégoire Détrez. 2012. Controlled Language for Everyday Use: The MOLTO Phrasebook. In *Proceedings of the Second International Conference on Controlled Natural Language*, CNL'10, pages 115–136, Berlin, Heidelberg. Springer-Verlag.

[Ranta2004] Aarne Ranta. 2004. Grammatical Framework. *Journal of Functional Programming*, 14(2):145–189.

[Ranta2009] Aarne Ranta. 2009. The GF Resource Grammar Library. *Linguistics in Language Technology*, 2.

[Ranta2011] Aarne Ranta. 2011. *Grammatical Framework: Programming with Multilingual Grammars*. CSLI Publications.

[Seki et al.1991] Hiroyuki Seki, Takashi Matsumura, Mamoru Fujii, and Tadao Kasami. 1991. On Multiple Context-Free Grammars. *Theoretical Computer Science*, 88(2):191–229.

[Stabler1997] Edward P. Stabler. 1997. Derivational Minimalism. In Christian Retoré, editor, *Logical Aspects of Computational Linguistics*, volume 1328 of *Lecture Notes in Computer Science*, pages 68–95. Springer, Berlin.

[Steedman1988] Mark Steedman. 1988. Combinators and grammars. In *Categorial Grammars and Natural Language Structures*, pages 417–442.